Programowanie z dowodzeniem twierdzeń

Piotr Masłowski

0 Słowem wstępu

Poniższe „zadania" kwalifikacyjne składają się w większości z wyjaśnień, opisów i kilku definicji. Ćwiczenia do zrobienia mają na celu przede wszystkim upewnić się, że pozostały materiał był zrozumiały i przystępny. Część z nich jest interaktywna, co – mam nadzieję – pozwoli rozwiać większość wątpliwości, czy błędnych interpretacji.

Zadania do wykonania luźno wzorowane są na zadaniach kwalifikacyjnych warsztatów z programowania funkcyjnego (Radosław Rowicki, 2019). Testowałem ten sposób wprowadzania rachunku lambda na kilku osobach (na żywo) w roku 2022 i komentarze które wtedy usłyszałem, to że „to takie trochę łamigłówki" oraz „ale to fajne", co pokrywa się z moim własnym doświadczeniem. Tak więc, polecam i pozdrawiam.

1 Lambdy (bez typów)

Rachunek lambda to prosty system formalny pozwalający budować i przekształcać wyrażenia reprezentujące funkcje anonimowe. Przykładowo $\lambda x.\ x$ oznacza funkcję przyjmującą jeden argument i od razu go zwracającą. Z kolei, $\lambda x.\ 3$ oznaczałoby funkcję również biorącą jeden argument, ale zawsze zwracającą 3, $\lambda x.\ 1 + x$ – funkcję dodającą 1, a $\lambda x.\ \cos\,(\sin\ x)$ – złożenie cosinusa z sinusem.

Co z takimi funkcjami możemy teraz zrobić? Możemy je zaaplikować do jakichś argumentów!

$$(\lambda x.\ x)\ 2 \to 2 \tag{1a}$$
$$(\lambda x.\ 3)\ b \to 3 \tag{1b}$$
$$(\lambda x.\ 1 + x)\ 5 \to 6 \tag{1c}$$
$$(\lambda x.\ \cos\,(\sin\ x))\ \pi \to 1 \tag{1d}$$

Z grubsza odpowiada to:

$$\mathrm{id}(x) := x \qquad \mathrm{id}(2) = 2 \tag{2a}$$
$$\mathrm{f}(x) := 3 \qquad \mathrm{f}(b) = 3 \tag{2b}$$
$$\mathrm{g}(x) := 1 + x \qquad \mathrm{g}(5) = 6 \tag{2c}$$
$$\mathrm{h} := \cos \circ \sin \qquad \mathrm{h}(\pi) = 1 \tag{2d}$$

Warto od razu zwrócić uwagę, że aby zaaplikować jakąś funkcję – czy to anonimową, czy nazwaną – nie musimy używać nawiasów. (m.in. jest dzięki temu czytelniej)

1.1 Formalizmy

Tyle intuicji na razie wystarczy. Teraz pora na definicje!
Niech X będzie zbiorem zmiennych (zwykłych oznaczeń, czy też napisów). Przykładowo $\{f, g, h, foo, bar, baz, x, y, z, a, b, c\}$.

Wyrażenia lambda są napisami tworzonymi w następujący sposób:

- Jeżeli $e \in X$, e jest wyrażeniem lambda.
- Jeżeli M jest wyrażeniem lambda oraz $e \in X$, to $(\lambda e.\, M)$ jest wyrażeniem lambda.
  (lambda-abstrakcja)
- Jeżeli M i N są wyrażeniami lambda, to $(M\ N)$ jest wyrażeniem lambda.
  (aplikacja funkcji)

Wyrażenia lambda możemy przekształcać następująco:

- alfa-konwersja:
  $$(\lambda e.\, M) \rightarrow (\lambda d.\, N)$$
  gdzie N to wyrażenie lambda otrzymane przez zastąpienie wszystkich wolnych wystąpień e w M, przez zmienną d, która nie ma wolnych wystąpień w M.

  (tutaj chodzi tylko o zmianę nazwy zmiennej)
- beta-redukcja:
  $$((\lambda e.\, M)\ N) \rightarrow O$$
  gdzie O to wyrażenie lambda otrzymane przez zastąpienie wszystkich wolnych wystąpień e w M przez wyrażenie N.

  (jest to po prostu mechaniczny sposób na aplikowanie funkcji)
- eta-konwersja:
  $$(\lambda e.\, (M\ e)) \rightarrow M$$
  oraz
  $$M \rightarrow (\lambda e.\, (M\ e))$$
  gdzie e nie ma wolnych wystąpień w M.

  (dodanie lub usunięcie lambda-abstrakcji i aplikacji dużo nie zmienia)

O co chodzi z tymi wolnymi wystąpieniami? Przykładowo, gdy weźmiemy wyrażenie lambda takie jak $\lambda x.\,((foo\,(\lambda x.\,x))\,x)$, potrzeba jakiegoś sposobu na odróżnienie który x jest od której lambdy.

$\lambda x.\,((foo\,(\lambda x.\,x))\,x)$	wszystkie x związane
$(foo\,(\lambda x.\,x))\,x$	jeden x związany, drugi wolny
$foo\,(\lambda x.\,x)$	jeden x związany
$\lambda x.\,x$	jeden x związany
x	jeden x wolny

Warto zauważyć, że tutaj foo jest aplikowane na $(\lambda x.\,x)$, a całe $(foo\,(\lambda x.\,x))$ – na x.

(Tak, tak. W powyższej definicji β-redukcji brakuje wstawki o tym, że N nie może mieć żadnych wolno występujących zmiennych, które zmieniałyby się w zmienne związane wewnątrz M/O.)

Co bardziej spostrzegawczy czytelnicy mogli już zauważyć, że nigdzie w definicjach nie pojawiły się wcześniej użyte liczby, sinusy czy arytmetyka. I rzeczywiście – niczego takiego nie zdefiniowaliśmy. Moglibyśmy oczywiście dodatkowo założyć istnienie tego wszystkiego. Zróbmy jednak coś ciekawszego: zbudujmy sobie matematykę na nowo.

1.2 Rozgrzewka

1.2.1 Zadanie 0 [mechaniczne; 6p] – redukcje i konwersje

Uprość najbardziej jak się da poniższe wyrażenia lambda (każda linia to osobny przykład) przy użyciu α-konwersji, β-redukcji i η-konwersji:
(wypisz kolejne kroki i które przekształcenia tam zaszły)

```
((λx.x) (λy.y)) (λz.z)
(λx.x) ((λy.y) (λz.z))
(λf.f x) (λy.g y)
(λg.((λf.f x) (λy.g y))) h
(λx.(λg.((λf.f x) (λy.g y))) h) z
(((λa.(λf.(λb.(f a) b))) (λg.(λy.g y))) (λc.c)) (λz.z)
```

1.3 Zabawa bez ograniczeń (czyli typów :P)

Na wstępie takie dwie uwagi:

1. Nawiasy i spacje nie są takie krytyczne. Domyślcie się co gdzie i jak. (Przede wszystkim jak jest $\lambda x.$ ..., to ... kończy się najdalej jak się da – argumenty mogą być dopiero za nawiasem)
2. Własne definicje pomocnicze są bardzo przydatne. Mimo że nie są częścią rachunku lambda, zachęcam do używania wszędzie gdzie zwiększają czytelność.

- wszystko na raz:

  ```
  λx. (((λy.y) (λy.y)) x)
  ```

- z definicją pomocniczą:

  ```
  id = λy.y
  λx. ((id id) x)
  ```

Niemniej, nie możemy tworzyć sobie w ten sposób nowej funkcjonalności – np. rekurencyjnych czy nieskończonych wyrażeń lambda!

Przejdźmy więc do zabawy

1.3.1 Zadanie 1a [nietrywialne; 2p] – wiele argumentów

Jak dotąd, wszystkie lambdy brały zawsze dokładnie jeden argument na raz, *no nie?* Jak więc dać jakiejś funkcji dwa argumenty?

Normalną odpowiedzią w matematyce byłyby pary. Takiego czegoś tutaj jednak nie mamy. Istnieją tylko zmienne, lambda-abstrakcje (z jednym parameterm) i aplikacje (również z jednym).

Żeby nie było to aż tak nieuchwytne, załóżmy że chcemy napisać wyrażenie lambda, które w jakiś sposób weźmie dwa argumenty – x i f – i zwróci ich aplikację – wyrażenie $f\ x$.

Do przetestowania swojego rozwiązania, zaaplikuj je na argumentach *bar* i foo.

Do sprawdzania wyników możecie użyć kalkulatora online: https://crypto.stanford.edu/~blynn/lambda/ Dostępne tam przykłady nic wam nie dadzą, bo używają trochę innej składni oraz są polukrowane – więc nie sugerujcie się nimi. (btw składnia taka jak tu – lambdy i kropki – również tam działa)

Rozwiązanie: (nie macie się nad tym męczyć)

Zamiana par, trójek, itd. w funkcjach na taki styl brania argumentów nazywana jest curryingiem – on nazwiska matematyka i logika, Haskella Curry'ego.

1.3.2 Zadanie 1b [3p] – złożenie funkcji

Napisz funkcję, która weźmie dwa wyrażenia lambda i dokona ich złożenia – zwróci funkcję która po zaaplikowaniu na jakimkolwiek wyrażeniu, zaaplikuje na nim drugą funkcję, a na wyniku pierwszą.

Testy:

```
złożenie foo bar         -- zwraca: λx. foo (bar x)
złożenie g f x           -- zwraca: g (f x)
złożenie (λx.x) (λx.h) x -- zwraca: h
```

Prostszy zapis:

- Gdy mamy kilka lambd, jedna w drugiej, możemy trochę ograniczyć pisanie nawiasów: $\lambda x\ y\ z.\ M$ oznacza $\lambda x.\ (\lambda y.\ (\lambda z.\ M))$
- Gdy mamy kilka aplikacji, jedna wokół drugiej, możemy ograniczyć pisanie nawiasów: $f\ x\ y\ z$ oznacza $((f\ x)\ y)\ z$

1.3.3 Zadanie 2a [nietrywialne; 2p] – prawda i fałsz

Budowanie matematyki zaczniemy od konstrukcji logiki boolowskiej. Na początek potrzebujemy prawdy i fałszu – dwóch wartości o bardzo podobnej strukturze, ale zasadniczo odróżniającym się od siebie nawzajem zachowaniu. Oczywiście, jak wszystko tutaj, muszą one być wyrażeniami lambda.

Spróbuj wymyślić takie dwa wyrażenia

Rozwiązanie: (również nie macie się nad tym męczyć)

Są to oczywiście tylko wyrażenia (napisy), które reprezentują (kodują) Boolowskie wartości logiczne. Możemy na nich dokonywać obliczeń, ale generalnie nie wystarczą one do przeprowadzania dowodów

1.3.4 Zadanie 2b [2p] – if-then-else

Napisz funkcję, która weźmie wartość logiczną oraz dwa inne argument i gdy ta wartość logiczna jest prawdą – zwróci pierwszy, a gdy fałszem – drugi.

Testy:

```
if prawda foo    bar   -- zwraca: foo
if fałsz  foo    bar   -- zwraca: bar
if fałsz  prawda fałsz -- zwraca: fałsz
```

1.3.5 Zadanie 2c [6p] – funkcje logiczne

Zaimplementuj `not`, `and`, `or`, `xor`, implikację i równoważność.

Testy:

```
not fałsz  -- zwraca: prawda
not prawda -- zwraca: fałsz

and prawda prawda -- zwraca: prawda
and prawda fałsz  -- zwraca: fałsz

xor prawda fałsz  -- zwraca: prawda
xor prawda prawda -- zwraca: fałsz
```

1.3.6 Zadanie 3a [lekko nietrywialne; 2p] – liczby naturalne

Żeby mieć liczby naturalne, potrzebujemy *całej serii* wyrażeń lambda o bardzo podobnej strukturze. (*Podpowiedź:* `0 = λf x. x`) Wymyśl co dalej.

Rozwiązanie: (nad tym również nie macie się męczyć)

1.3.7 Zadanie 3b [4p] – arytmetyka

Zaimplementuj następnik (funkcję dodającą 1), dodawanie, mnożenie oraz potęgowanie.

Testy:

```
succ 0 -- zwraca: 1
succ 4 -- zwraca: 5

plus 2 3 -- zwraca: 5
plus 0 0 -- zwraca: 0

razy 2 3 -- zwraca: 6
razy 6 1 -- zwraca: 6
razy 7 0 -- zwraca: 0

potęga 2 3 -- zwraca: 8
potęga 5 0 -- zwraca: 1
```

```
potęga 0 5 -- zwraca: 0
```

1.3.8 Zadanie 3c [nietrywialne; 6p] – trudna arytmetyka

Zaimplementuj poprzednik (funkcję odejmującą 1) oraz odejmowanie. Zamiast liczb ujemnych, zawsze zwracaj 0.

Testy:

```
pred 5 -- zwraca: 4
pred 1 -- zwraca: 0
pred 0 -- zwraca: 0

minus 2 3 -- zwraca: 0
minus 7 2 -- zwraca: 5
minus 9 9 -- zwraca: 0
```

Jeżeli brakuje ci pomysłów, zostaw to zadanie na po następnym (albo po jeszcze kolejnym).

1.3.9 Zadanie 4a [3p] – pary

Przy pierwszym zadaniu było powiedziane, że nie mamy par. Ale to nie znaczy, że nie możemy sami ich sobie zrobić!

```
para = λx y. λf. f x y
```

0. Napisz funkcję, która wyjmie pierwszy element (`x`) z dowolnej pary.
1. Napisz funkcję, która wyjmie drugi element (`y`) z dowolnej pary.
2. Elegancko dodaj liczby zawarte w `para 5 3`.

Testy:

```
przypadek0 = para 4 2
przypadek1 = para 0 prawda
przypadek2 = para 5 3

??? -- zwraca: 4
??? -- zwraca: prawda
??? -- zwraca: 8
```

1.3.10 Zadanie 4b [5p] – unie

Przeciwieństwem par, są unie. Gdy tworzymy parę, potrzebujemy dwóch wartości, a później możemy wyciągnąć z pary dowolną z nich. Unię możemy stworzyć na dwa sposoby – na `lewo` i na `prawo`. Z każdego z nich możemy wyciągnąć wartość która użyta była przy konstrukcji. Na dodatek, zwykle nie wiemy później na który sposób unia była stworzona, więc musimy obsłużyć oba przypadki.

Napisz dwie funkcje (`lewo` i `prawo`), obie przyjmujące po jednej wartości, które zwrócą funkcję biorącą dwie funkcje – pierwszą na wypadek gdyby oryginalnie użyte było `lewo`, a drugą na wypadek gdyby oryginalnie użyte było `prawo`. Ostatecznym wynikiem powinna być aplikacja odpowiedniej z podanych funkcji na wartości podanej na samym początku.

Brzmi to dość zagmatwanie, ale mam nadzieję, że przykłady wszystko wyjaśniają:

Testy:

```
przypadek0 = lewo 3
przypadek1 = prawo fałsz
przypadek2 = prawo prawda
przypadek3 = lewo 0

przypadek0 succ not -- zwraca: 4
przypadek1 succ not -- zwraca: prawda
przypadek2 succ not -- zwraca: fałsz
przypadek3 succ not -- zwraca: 1
```

2 Typy proste

No to lambdy już znamy. Teraz czas na typy!

Ale po co ktokolwiek je dodawał? Okazuje się, że niektóre wyrażenia lambda moglibyśmy redukować bez końca i wciąż nie osiągnęłyby żadnej ostatecznej formy.

Przykładowo, tzw. kombinator Y:

$$Y = \lambda f.\ (\lambda x.\ f\ (x\ x))(\lambda x.\ f\ (x\ x))$$

ma tą właściwość, że $Y\ f$ jest β-równoważne $f\ (Y\ f)$.

$$\begin{aligned} Yg &= (\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))g \\ &\overset{\beta}{\rightarrow}(\lambda x.g(xx))(\lambda x.g(xx)) \\ &\overset{\beta}{\rightarrow} g((\lambda x.g(xx))(\lambda x.g(xx))) \\ &\overset{\beta}{\leftarrow} g((\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))g) \\ &= g(Yg) \end{aligned}$$

Dodając typy, możemy zapewnić tzw. silną normalizację. Oznacza to, że dowolna sekwencja przepisywania (redukcji i konwersji) prędzej czy później osiągnie postać normalną. Ale czym właściwie są te typy? Zacznijmy od definicji.

2.1 Formalizmy

Niech X będzie zbiorem zmiennych, a T będzie zbiorem typów podstawowych. Wtedy $e : \tau$, gdzie $e \in X$ i $\tau \in T$, oznacza że e jest typu τ. Przykładowo:

$$X = \{f, g, h, foo, bar, baz, x, y, z, a, b, c\}$$
$$T = \{A, B, C\}$$
$$x : A$$

Typy wyrażeń lambda są napisami tworzonymi w następujący sposób:

- Jeżeli $\tau \in T$, τ jest typem.
- Jeżeli τ i σ są typami, $\tau \to \sigma$ też jest typem.

Otypowane wyrażenia lambda są napisami tworzonymi w następujący sposób:

- Jeżeli $e \in X$ i $e : \tau$, to e jest wyrażeniem lambda typu τ.
- Jeżeli M jest wyrażeniem lambda typu σ, oraz $e \in X$ i $e : \tau$, to $(\lambda e : \tau . M)$ jest wyrażeniem lambda typu $\tau \to \sigma$.
  (lambda-abstrakcja)
- Jeżeli M i N są wyrażeniami lambda typów $\tau \to \sigma$ i τ odpowiednio, to $(M\ N)$ jest wyrażeniem lambda typu σ.
  (aplikacja funkcji)

Wtedy $M : \tau$ oznacza, że M jest wyrażeniem lambda typu τ.

Reguły przekształcania otypowanych wyrażeń lambda niewiele różnią się od tych bez typów:

- alfa-konwersja:
  $$(\lambda e : \tau .\ M) \to (\lambda d : \tau .\ N)$$
  gdzie $N : \sigma$ to wyrażenie lambda otrzymane przez zastąpienie wszystkich wolnych wystąpień e w $M : \sigma$, przez zmienną $d : \tau$, która nie ma wolnych wystąpień w M.
- beta-redukcja:
  $$((\lambda e : \tau .\ M)\ N) \to O$$
  gdzie $O : \sigma$ to wyrażenie lambda otrzymane przez zastąpienie wszystkich wolnych wystąpień e w $M : \sigma$ przez wyrażenie $N : \tau$.
- eta-konwersja:
  $$(\lambda e : \tau .\ (M\ e)) \to M$$
  oraz
  $$M \to (\lambda e : \tau .\ (M\ e))$$
  gdzie e nie ma wolnych wystąpień w $M : \tau \to \sigma$.

2.2 Aqua aerobik — bo dużo się tu nie będzie działo

2.2.1 Zadanie 5a [mechaniczne; 2p] – redukcje i konwersje

Uprość najbardziej jak się da poniższe wyrażenia lambda przy użyciu α-konwersji, β-redukcji i η-konwersji:
(wypisz kolejne kroki i które przekształcenia tam zaszły)

```
(λx:(A->A). x) ((λy:(A->A). y) (λz:A. z))
```

```
x : A
g : A->B
(λf:(A->B). f x) (λy:A. g y)
```

2.2.2 Zadanie 5b [3p] – typowanie

Otypuj poniższe wyrażenia – znajdź takie dowolne przypisanie typów do zmiennych, żeby poniższe beztypowe wyrażenia lambda stały się otypowanymi wyrażeniami lambda. Jeżeli jest to niemożliwe, napisz.

```
λf x. f (f x)
λf x. f (f x) x
(λx. x) (λy. y) z
(λx. x) (λy. y) (λz. z)
(λg. (λf. f x) (λy. g y)) h
(λx. (λg. (λf. f x) (λy. g y)) h) z
```

2.2.3 Zadanie 6 [1p] – złożenie funkcji

Napisz funkcję dokonującą złożenia trzech funkcji o typach $A \to B$, $B \to C$ i $C \to D$.

Prostszy zapis:

- Typy takie jak $A \to (B \to (C \to D))$ możemy zapisywać jako $A \to B \to C \to D$

2.2.4 Zadanie 7 [2p] – moja prawda, twoja prawda

Spróbuj otypować wartości logiczne. Jakie widzisz z tym problemy? Czy jest to możliwe?

2.2.5 Zadanie 8 [5p] – liczby i arytmetyka

Spróbuj otypować liczby naturalne oraz funkcje takie jak następnik i poprzednik.

2.2.6 Zadanie 9a [3p] – typy par

Spróbuj otypować pary i funkcje na nich.

2.2.7 Zadanie 9b [2p] – typy unii

Spróbuj otypować unie i funkcje na nich.

3 System F

Jak przekonaliśmy się w poprzedniej sekcji, wiele konstruktów w rachunku lambda przestało być możliwych po dodaniu typów. Inne z kolei, były zduplikowane pomiędzy różnymi typami. Nawet tak prosta funkcja jak identyczność $\lambda x.\, x$, nagle otrzymała niezliczone warianty: $\lambda x : A.\, x$, $\lambda x : B.\, x$, $\lambda x : C.\, x$, $\lambda x : D.\, x$, itd.

Problem par i unii można rozwiązać poprzez zdefiniowanie nowych konstruktorów typów $\tau \times \sigma$ oraz $\tau + \sigma$ – do kolekcji z $\tau \to \sigma$. Wymaga to również dodania projekcji – funkcji dekonstruujących wspomniane pary i unie – istniejących dla wszystkich możliwych typów.

Bardziej ogólnym rozwiązaniem są tak zwane duże lambdy. Ta idea polega na wprowadzeniu drugiej lambda-abstrakcji, która różni się od tej zwykłej tym, że zamiast wartości, bieże typy. Przykładowo, identyczność można wtedy zapisać jako $\Lambda t.\, \lambda x : t.\, x$. Następnie, gdy chce się jej użyć dla jakiegoś konkretnego typu, wystarczy ją na nim zaaplikować:

$$(\Lambda t.\, \lambda x : t.\, x) A \to \lambda x : A.\, x$$

Nazywane jest to również „polimorficzny rachunek lambda" albo System F.

Ale jaki typ ma taka duża lambda? W rachunku lambda z typami prostymi, funkcje mają typy postaci $\tau \to \sigma$. Tutaj więc, potrzebowalibyśmy jeszcze jakiejś „dużej strzałki". I rzeczywiście, System F definiuje nowy konstruktor typów.

$$(\Lambda t.\, \lambda x : t.\, x) : \forall t.\, t \to t$$

3.1 Powtarzalna zabawa

3.1.1 Zadanie 10 [3p] – złożenie funkcji

Napisz funkcję dokonującą złożenia funkcji o dowolnych typach. Następnie zaaplikuj ją na $f : E \to F$ oraz $g : D \to E$. Zredukuj wynik.

3.1.2 Zadanie 11 [4p] – logika i liczby

Napisz polimorficzne wersje wartości logicznych, liczb, funkcji `and` oraz następnika.

3.1.3 Zadanie 12a [3p] – pary

Skonstruuj polimorficzne pary, oraz projekcje – `pierwszy` i `drugi`.

3.1.4 Zadanie 12b [2p] – unie

Skonstruuj polimorficzne unie i pokaż jak się ich używa.

4 System F_ω

Jak się dobrze przyjrzeć, to ten typ dużych lambd, sam wygląda trochę jak lambda. Nazwa która po nim stoi, jest zmienną – argumentem funkcji. Co by się więc stało, gdyby faktycznie wprowadzić funkcje na poziomie typów? To znaczy, zacząć używać lambda-abstrakcji oraz aplikacji przy konstrukcji samych typów – a nie tylko wyrażeń. Co więcej, wtedy można by przypisać typy typom, oraz dokonać tego samego również na tamtym poziomie. Pociągnięcie tej idei w nieskończoność, nazywa się System F_ω.

Jakie są tego zastosowania? Jesteśmy teraz w stanie tworzyć własne konstuktory typów. Na przykład pary można teraz zdefiniować jako:

$$\begin{aligned}\mathrm{Para} &= \lambda a : \mathrm{Typ}.\ \lambda b : \mathrm{Typ}.\ \forall c.\ (a \to b \to c) \to c \\ \mathrm{para} &: \forall a.\ \forall b.\ a \to b \to \mathrm{Para}\ a\ b\end{aligned}$$

5 Π – typy zależne

Czy jest w ogóle możliwe pójść dalej? Osiągnęliśmy przecież już nieskończoność. Otóż tak. Jeżeli dobrze się przyjrzymy jakie lambdy mamy dostępne, zauważymy, że są w naszym arsenale lambdy które:

1. biorą wartości i zwracają wartości
2. biorą typy i zwracają typy
3. biorą typy i zwracają wartości

Pozostają więc jeszcze lambdy które brałyby wartości, a zwracały typy.

Nowa lambda wymaga nowego typu. Podobnie jak typ dużych lambd, ten będziemy również oznaczać przy użyciu „kwantyfikatora”:

$$\Pi x : A.\ Foo\ x\ B$$

(Tutaj $Foo : A \to \mathrm{Typ} \to \mathrm{Typ}$ jest funkcją biorącą jedną wartość typu A oraz jeden typ.)

Ale czy takie lambdy byłyby w ogóle przydatne? Tak! Przykładowo, możemy dzięki nich mówić o polimorficznych typach, sparametryzowanych liczbami. Oto typ funkcji biorącej wektor (listę z określoną długością) i doklejającej do niego dokładnie jeden element:

$$\Pi n : \mathrm{Int}.\ \forall a.\ a \to (\mathrm{Vect}\ n\ a) \to (\mathrm{Vect}\ (n+1)\ a)$$